The SJ Framework for Transport-Independent, Type-Safe, Object-Oriented Communications Programming

Raymond Hu Nobuko Yoshida Andi Bejleri Kohei Honda

Imperial College London Queen Mary, University of London

Abstract

Communications programming, involving complex message exchanges over multiple transports, is an omnipresent element in modern distributed applications. Existing engineering solutions however have considerable limitations: there is no portability across differing transports. Programming abstractions for communication are typically provided through low-level APIs, bound to specific transports or application domains, without offering either type or protocol safety.

This paper proposes an extensible Java-based language and runtime framework which enables safe and efficient virtualisation of communications programming across heterogeneous transports. Application programmers describe communications in terms of high-level, structured sessions, without concern for underlying transport mechanisms. After type-checking, the compiler generates a transport-independent intermediate form, which can be efficiently executed by the runtime across different transports whilst ensuring communication safety. Through portable, abstract low-level communication instructions defined by the *Abstract Transport*, a new transport can quickly and seamlessly integrate with existing session services. A case for transport independence is made using concrete applications from widely different domains, including parallel algorithms, a Web-based application server, and Internet chat. The benchmark results show this framework imparts significant gains in portability, safety and productivity, as well as efficient utilisation of individual transports through type-directed optimisation.

1. Introduction

Programming for multi-transport environments. Communication is an increasingly fundamental element in programming for an extensive range of application domains. These include programs communicating within and between SMP nodes in high-performance clusters; message exchange across global financial networks; and applications integrating services distributed across the Internet and the Web. Many of these applications exhibit complex, application-specific interaction patterns that involve dynamic changes to communication topology, e.g. the migration of an on-going conversation to another process. Moreover, this interaction can span over various transports with differing characteristics, and even different protocol layers (e.g. TCP and HTTP).

VM platforms for object-oriented programming such as the JVM and CLR provide *hardware/OS independence* and *type-safety*. By abstracting away machine and OS details, such platforms enable the portable execution of high-level programs. Distributed applications, however, do not enjoy a similar portability for communication across the diverse transports employed in modern computing environments. The separate application components may take advantage of local machine virtualisation, but inter-component interaction often relies on non-portable, low-level communication APIs or domain-specific middleware.

Firstly, the majority of networking APIs (e.g. sockets) implicitly couple programs to specific transports, hence exposing all the functional details of these transports to the application programmer. Working with raw byte streams/packets without type-safety, whilst flexible, is too low-level: there is no clear separation of the application-level interaction from the underlying communication mechanisms. Secondly, many communications APIs, including messaging middleware, are designed for specific application domains, offering communication functions useful for specific application tasks (e.g. corporate messaging, parallel programming, P2P), but not general abstraction for portable communications programming [26, 27, 31]. Thirdly, RPC-based APIs such as Java RMI, CORBA and their XML variants, despite offering type-safety, blanketly restrict communication to synchronous call-return patterns, which is too rigid for capturing a variety of interaction structures. Call-return can also limit the effective use of transport functionality, e.g. for properties tied to connection, such as security through SSL/TSL. Thus, for applications that demand flexibility, the developer must be concerned with the details of specific transports, and the resulting code is not portable across different network environments.

These issues severely impair not only productivity, but also the potential for formal safety assurance, and the speedy and effective exploitation of existing and new transports.

Transport independence. Against this background, the present paper argues for the significance of portable communications programming in object-oriented languages through *transport independence*. In transport-independent programming, the programmer describes communications through high-level, type-safe programming abstractions decoupled from individual transports, just as programming in high-level object-oriented languages such as Java and C♯ is decoupled from local hardware/OS details. The resulting description should then be executable through interactions between the *communication runtimes* at the two ends over a variety of network transports, just as the JVM and CLR support local execution of machine-neutral intermediate code over a variety of hardware and operating systems.

Transport independence demands both language facility for flexible, type-safe communications programming suited to all major application domains; and a runtime design that enables effective exploitation of diverse transports. Runtime extensibility, through the integration of new and specialised transports, is also essential for supporting application-specific optimisation and for the adoption of rapidly evolving networking technologies. Our argument for the feasibility of transport-independent programming derives from our observation that these goals can be achieved by a language-runtime design centring on an abstraction for communications, *session* [14, 22, 42].

A session can be viewed as an application-level abstraction for *connection*, as found in e.g. TCP, SMTP and DCCP [28], which is a fundamental idea in network engineering across diverse protocols [43]. A connection gives a unit of flow control and QoS assurance: any widely used transport requires some form of flow control (cf. [28]), demanding a delineation of a beginning and an end of message exchanges so that the protocol can (enumerate and) control these messages, giving rise to the notion of connection [43].

A session reifies this idea as a high-level, type-safe programming abstraction, through language constructs and a type theory. It organises communications into logical units of structured conversations, or *sessions*, and offers an efficient static validation through *session types*. The validation assures both type and protocol safety — not only is the value of each message correctly typed, but messages in a session are exchanged according to the scenario specified by the associated session type, precluding communication mismatch, a typical bug in communications programming. In this paper we shall explore the role of session both as a natural abstraction for communications and as a basis of effective usage of transports.

Use cases for transport independence. We substantiate our vision of transport independence through typical scenarios in communications programming. We start from a field where communications are needed for performance.

> ***Use case 1: parallel algorithms.*** *A message-passing parallel algorithm implementation needs to be executed in (and across) varied computing environments whilst using available transports effectively, e.g. TCP (in LAN), RDMA (in HP clusters), and shared memory in SMP machines and multicore CPUs.*

Here, both portability *and* efficiency are important: we require the communicating processes to not only run over diverse transports but to also make the most of them. Further, parallel algorithms often exhibit linear data usage, which opens the potential for optimisation through suitable language support.

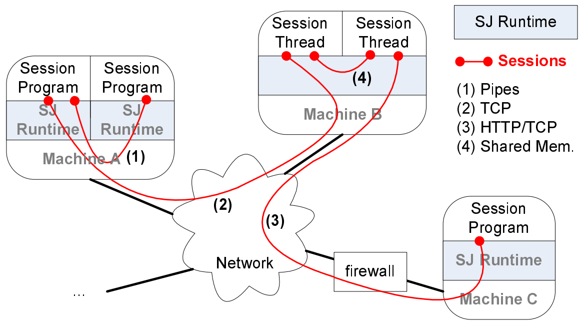

Figure 1. Sessions across heterogeneous nodes/transports.

Next we consider a typical distributed application, instant messaging over the Internet.

> ***Use case 2: Internet chat.*** *Clients can communicate via a chat server and also through direct connections. Transport availability between the clients and server may be restricted by e.g. firewalls; clients may also impose additional security requirements. Hence, a variety of transports (TCP, SSL/TSL, HTTP-tunnelling), are required to meet application functionality.*

Note that the message exchanges in a chat conversation naturally form a session, which should be independent from specific transports and physical connections. Such applications strongly motivate the decoupling of general purpose communications programming from the underlying transports.

A final use case demonstrates an advanced application of transport independence.

> ***Use case 3: session-based Web services.*** *A client contacts a Web server through HTTP to start an application session. The request is delegated to an application server, and the session continues between the client and the application server using e.g. TCP, SSL.*

This is an instance of *cross-transport session migration*, where a session endpoint is transparently migrated from one peer to another across different transports, an interaction pattern that arises naturally in many application scenarios.

Although these use cases are taken from significantly different domains, all share the need for high-level abstraction from not only the local machine, but also the *network*: we need a means for ***type-safe communications programming over heterogeneous network nodes and transports***. Figure 1 depicts a general scenario for collaborative tasks involving concurrent and distributed sessions. These sessions, implemented through transport-independent session programming, are now running over the different transports available in (moreover, most suited to) each context: sessions between threads and local processes use shared memory and pipes, whilst distributed peers may use TCP, backed up by HTTP/TCP for e.g. firewall traversal. Similarly, session delegation between these peers automatically adjusts for transport binding. The key is that transport independence realises this scenario whilst freeing the application programmer from the manipulation of raw transports and connections.

Challenges for transport independence. This paper presents design, implementation and usage experiences on a language-runtime framework that extends Java for transport-independent object-oriented communications programming. We examine the feasibility of such a framework from the requirements of portability, safety and performance. Our work complements other elements of distribution, such as object migration [7], to focus on the key technical challenges for realising transport independence. These include:

1. (Programming) Can we provide a practically expressive and type-safe programming framework for a wide range of concurrent and distributed applications?
2. (Transport Usage) Can we design a language-runtime framework which can make the most of the performance and features of concrete transports, and which can effectively realise cross-transport functions (e.g. delegation)?
3. (Extensibility) Can we incorporate a new transport quickly and seamlessly? Can the language and runtime be extended to support significant transport-specific functionality whilst retaining consistent semantics for transport-independent communications programming?

In the subsequent sections we shall show how our framework, centring on session abstraction, meets these challenges. The language-runtime is built on an existing extension of Java with sessions, SJ [24]. A benefit of using Java is that portability and type safety of the local portion of programs are provided by Java typing and the JVM. In order to generalise these principles to the communication portion of concurrent and distributed programming, the present work extends SJ with new language constructs for expressive, transport-independent communications programming, integrated with an extensible runtime which enables efficient virtualisation of communications over multiple concrete transports. We refer to this language-runtime framework as the *SJ Framework*. To summarise, the main technical contributions of the present work include:

- A language and runtime framework that extends SJ for transport-independent session-based communications programming, including uniform alias control for objects and session typing (§3). The SJ framework readily supports a wide range of applications, illustrated through concrete implementations of the above ***Use cases*** (§3,4).
- A highly extensible runtime architecture that offers rich high-level interaction services for executing transport-independent programs, including type-safe cross-transport session migration with transparent negotiation of optimal transports. The current runtime implementation incorporates multiple concrete transports such as TCP, SSL, shared memory, HTTP and HTTPS (§5). A key design element is the *Abstract Transport*, which defines abstract, low-level communication instructions efficiently implementable by diverse transports (§5).
- An empirical evaluation of the framework through both micro and macro benchmarks. The results show the transport independent design incurs very little overhead over concrete transports whilst ensuring type safety (§6).

Additional contributions include new session programming features that give greater expressiveness, such as session-multicasting (§3.1), session-iteration chaining (§3.2), session-thread spawning, session-recursion and higher-order service passing (§4.1). The compiler and runtime, applications, and omitted benchmarks are available at [41].

2. The SJ Framework: Overview

The SJ Framework aims to provide a highly extensible platform for transport-independent object-oriented communications programming on the basis of session abstraction. The framework design follows the standard end-to-end principle in network engineering [36], where the required abstraction, session-based interaction, is realised by communication actions performed at the endpoints, the SJ Runtime instances running over JVMs. Unlike standard network layerings [43], sessions are *not* tied to a fixed network layer, since the same communication abstraction should be maintained over TCP, HTTP, DCCP or even a link-layer protocol.

To simultaneously attain efficiency and extensibility in this multi-transport setting, the SJ Framework places a thin layer of abstraction on top of each concrete transport. This abstraction, called *Abstract Transport*, specifies a set of portable low-level communication instructions, to be implemented by each concrete transport (see §5 for details). The semantics of language constructs for session programming is realised by *(session) interaction services*, which are in turn defined over the Abstract Transport, and thus decoupled from individual transports. This decoupling is essential for meeting the *extensibility* challenges in §1:

- A new service implemented over Abstract Transport instantly runs over all existing and future transports, without re-implementation for each transport.
- Symmetrically, a new transport can be seamlessly integrated by implementing the Abstract Transport, instantly available to all existing and future interaction services.

As an example, consider services which work across different transports, such as cross-transport session migration (***Use case 3*** in §1). Implementing such a service over concrete transports would inevitably increase the amount of plumbing required for each additional transport, resulting in error-prone, delayed deployment of the new transport.

Compilation and execution life cycle. Figure 2 depicts the compilation and execution stages of the SJ Framework, which operate across the layered architecture (session program, interaction services, abstract transport) discussed above. We briefly describe each layer below.

SJ Application Layer: The SJ Framework offers the application programmer a rich language facility for transport-independent object-oriented session programming. The SJ compiler, implemented using Polyglot [34], statically

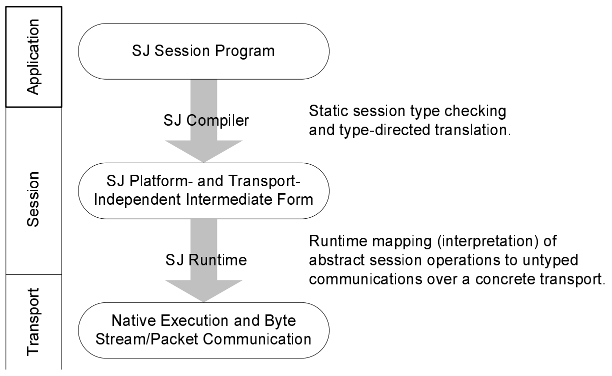
Application
Session
Transport
SJ Session Program
SJ Compiler
Static session type checking and type-directed translation.
SJ Platform- and Transport-Independent Intermediate Form
SJ Runtime
Runtime mapping (interpretation) of abstract session operations to untyped communications over a concrete transport.
Native Execution and Byte Stream/Packet Communication

Figure 2. Compilation-execution stages of SJ Framework.

type checks SJ programs, and generates a transport-independent Java-based intermediate form by translating session operations into calls to the interaction services.

SJ Session Layer The SJ Runtime has two main responsibilities. The first is performing the interaction services over the Abstract Transport. Services are incorporated into the SJ Runtime as SJ *service components*. Example services include session initiation (which validates session peer compatibility), the wire format and serialization for communicating session messages, and cross-transport session migration for delegation as mentioned above.

SJ Abstract Transport Layer The second role of the SJ Runtime is managing concrete connections, established using available transports, to realise the semantics of the Abstract Transport. The Abstract Transport operations are executed as actions on the underlying transports as directed by SJ *transport module* implementations. This layer also includes *transport negotiation* between the SJ Runtime instances of peers at session initiation, to agree on a transport for the new session based on the execution context and transport configuration parameters.

The following sections substantiate the effectiveness of this layered architecture from three different viewpoints. First, §3 and §4 illustrate the key elements of session-based transport-independent programming, through concrete implementations of the three use cases from §1. Then §5 presents highlights from the design and implementation of the SJ Runtime, including virtualisation mechanisms and the utilisation of type information. Finally, §6 reports performance results.

3. SJ for Parallel Algorithms

This section and the next illustrate the versatility of transport-independent session programming, demonstrating how it offers natural abstraction for communication in widely differing application domains. This section focuses on message-passing parallel algorithms, substantiating ***Use case 1*** in §1. Parallel algorithms use communication for performance gain, and can be characterised by their tightly-coupled, deterministic message-passing computation. We treat two representative parallel algorithms, a parallel implementation of the Jacobi Method for solving the Discrete Poisson Equation (referred to as “Jacobi”) and a standard simulation for the n-Body problem (“n-Body”). Along the way we introduce the basic elements of transport-independent session programming in the SJ Framework, including the use of *noalias types*. The integration of noalias types and session communication precisely captures linear data usage patterns typical in scientific computing, and enables transparent, type-directed runtime communication optimisations for such data. Performance results for these algorithms and comparisons with a Java implementation of MPI [1], are presented in §6.[1]

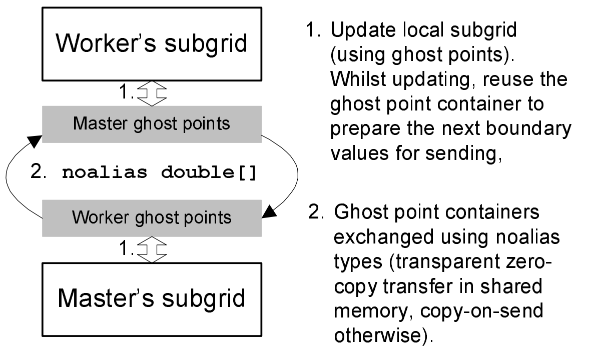
Worker's subgrid
1.
Master ghost points
2. noalias double[]
Worker ghost points
1.
Master's subgrid
1. Update local subgrid (using ghost points). Whilst updating, reuse the ghost point container to prepare the next boundary values for sending,
2. Ghost point containers exchanged using noalias types (transparent zero-copy transfer in shared memory, copy-on-send otherwise).

Figure 3. Master-Worker ghost points exchange.

```
protocol masterToWorker {
  cbegin. // Request the Worker service.
  !<int>. // Send matrix size.
  ![      // Enter main loop.
    !<double[]>. // Send our boundary values.
    ?(double[]). // Receive Worker ghost points.
    ?(double)    // Receive convergence data.
  ]*.      // After the last iteration...
  ?(double[][]) // ...receive the final results.
}
```

Figure 4. Session type for the Master-Worker interactions in the SJ implementation of the parallel Jacobi algorithm.

3.1 Jacobi Solution of the Discrete Poisson Equation

Poisson's equation is widely used in many areas of the natural sciences, including electrostatics, thermal dynamics, and climate computations. The discrete two-dimensional Poisson equation $(\nabla^2 u)_{ij}$ for a $m \times n$ grid can be written

$$u_{ij} = \frac{1}{4}(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - dx^2 g_{i,j})$$

where $2 \le i \le m-1$, $2 \le j \le n-1$, and $dx = 1/(n+1)$. Jacobi's method converges on a solution by repeatedly replacing each element of the matrix u by an adjusted average of its four neighbouring values. adjusted by $dx^2 g_{i,j}$; for this example, we set each $g_{i,j}$ to 0. Then from the k-th approximation of u, the next iteration calculates

[1] A preliminary summary of the algorithms in this section was presented in an informal online PLACES'09 workshop pre-proceedings (6 pages).

```
noalias double[] ghost1 = new double[size], ...;
... // Grid values and ghost points initialised.
<mw1, mw2>.outwhile( // Main Master loop: ![..]*.
      !converged(...) && iters++ < MAX_ITERS) {
  mw1.send(ghost1); // (⋆) mw1: !<noalias double[]>
  ... // ghost1 variable becomes null.
  ghost1 = mw1.receive(); // mw1: ?(double[])
  ... // Update subgrid and prepare next ghost points.
  convergenceData = mw1.receive(); // mw1: ?(double)
}
```

```
noalias double[] ghost = ..., prev = ...;
... // Grid values and ghost points initialised.
<wm>.inwhile() { // Worker follows Master: ?[..]*.
  prev = ghost;
  ghost = wm.receive(); // wm: ?(double[])
  ...
  wm.send(prev); // (⋆) wm: !<noalias double[]>
  ... // Subgrid and ghost points updated.
  wm.send(convergenceData); // wm: !<double[]>
}
```

Figure 5. SJ implementation of the main loop in the parallel Jacobi algorithm.

$$u_{ij}^{k+1} = \frac{1}{4}(u_{i+1,j}^k + u_{i-1,j}^k + u_{i,j+1}^k + u_{i,j-1}^k)$$

Termination may be on reaching a target convergence threshold or on completing a certain number of iterations.

Parallelisation of this algorithm comes from the ability to update each element independently (within one iteration). The grid can be divided so that each subgrid is processed in separate processes/threads. The processes with neighbouring subgrids need to interact to exchange their subgrid boundary values, and to reach a consensus for termination. The boundary values cached by each process are termed *ghost points*.

To focus on the points of interest, we consider a one-dimensional decomposition of a square grid into three non-overlapping subgrids (of any size) for three separate processes. The process managing the central subgrid is designated the *Master* (Figure 3), who controls the termination condition for all three processes; the two end processes are *Workers* who only directly interact with Master.

Jacobi protocols. Session programming starts from the declaration of protocols for the intended interaction using session types [24]. Figure 4 lists the session type for the Master-Worker interactions from the perspective of the former. Master (the client) requests a session (`cbegin`) with each Worker service. After the session is established, Master sends the matrix size (`!<int>`), from which Worker determines its subgrid size and initializes the grid values. Master and Worker then enter the main iteration loop (`[..]*`) of the algorithm, under the control of Master (`![`). In each iteration step, Master and Worker send and receive their updated subgrid boundary values (the ghost points: `!<double[]>.?(double[])`), and Master receives from Worker the convergence data for the Worker subgrid (`?(double).?(double)`). From the convergence data, Master decides to continue the iteration or to terminate, at which point Worker returns the final results of its complete subgrid (`?(double[][])`). Master performs the same interactions with each Worker.

Jacobi implementation: using noalias types. SJ session programming involves implementing the declared protocols using the session communication operations and interaction constructs, performed via session sockets (`SJSocket`s) [24]. The SJ compiler statically type checks session implementations (e.g. the extracts in Figure 5) against the associated session types (Figure 4), guaranteeing the correct interaction behaviour for the program.

As explained above, the Master and Workers exchange their subgrid boundary values, i.e. ghost-points, in each iteration. Figure 3 depicts the scheme where, in each iteration step, the new boundary values to be sent *next* iteration are copied into the current ghost points "container" (i.e. `double []`) as the update proceeds. The key point is that, after sending, the sender does not use this object again since a new ghost points array is received each iteration: the sender is in effect *giving away* this object, which satisfies the property of *unique ownership* [2]. In the SJ Framework, such variables can be declared through the `noalias` modifier. A noalias reference becomes null after being assigned and when passed as a method argument, which includes communication via session `send`. The SJ compiler enforces correct usage based on a standard typing approach to ensuring alias freedom [2]. This linear data usage is a common pattern in message-based parallel algorithms (for example, a similar treatment of communication data is found in the n-body simulation below).

Figure 5 extracts from the implementations of the main loops in Master and Worker. To synchronise with the Workers, Master uses the multicast session-iteration, `<mw1, mw2>.outwhile(...){...}`, where `mw1` and `mw2` are Master's session sockets to the two Workers; each Worker uses the dual construct (`inwhile`) to follow the control flow of Master. In Master, the noalias variable `ghost1` is used to hold the current ghost point array from Worker 1 (there would similarly be a `ghost2`). The communication of this variable, i.e. sending a noalias message (`!<noalias double[]>`), is marked ($\star$). The received ghost point arrays are implicitly noalias, and hence can be assigned to the noalias variables.

Noalias and transport independence. Noalias typing imposes a precise abstraction for non-shared objects with correct usage enforced through static checking, whilst inherently delivering the desired semantics for sending noalias messages. This language facility, due to the integration of object alias control and session communication, gives the freedom to optimise the communication of noalias messages *transparently*, regardless of the underlying transports:

1. If a session is executed in a shared memory context, noalias messages can be delivered by reference passing.

2. It facilitates asynchronous sending over synchronous transports, and those with limited ability to buffer outbound messages, by precluding unnecessary copying.

The first can be exploited for sessions running on e.g. a multicore machine, now a standard hardware configuration. As §5 discusses, the SJ Runtime (by default) arranges shared memory as a session transport between peers in such a configuration. Noalias messages can then be passed by reference to the receiver, precluding any copying or serialization. Otherwise, the SJR transparently falls back to copy-on-send: as described, the semantics of noalias argument passing (the argument becomes null) maintains consistent semantics for noalias message passing in either case. The second optimisation will also be effective if the underlying transport supports e.g. RDMA-like communication mechanisms.

The SJ Framework uniformly integrates noalias typing, as a general purpose alias control mechanism, with session abstraction. In addition to semantic transparency for noalias communication over differing transports, session typing utilises noalias properties to contribute towards ensuring session linearity [14, 22], giving a natural treatment of session delegation (§4.2). Other middlewares for high-level communication, like CORBA, may offer zero-copy transfer as a runtime configuration setting, but only by altering program semantics. In Sing♯ [17] (used for Singularity OS) which first presented a similar use of linear messaging over shared memory for systems-level programming, messages are restricted to scalar values coupled to a lower-level, transport-specific mechanism, linear heap exchange (see §8).

3.2 Simulation of the n-Body Problem

The n-Body problem involves finding the motion, according to classical mechanics, of a system of bodies (particles with mass) given their initial position and velocities. The following SJ implementation demonstrates how a complex collaboration pattern involving an arbitrary number of agents can be naturally implemented using typed sessions, taking advantage of noalias types and requiring the new SJ constructs for multiparty session-iteration chaining. Moreover, portability due to transport independence permits highly flexible deployment for this single implementation: the agents can be co-process threads, co-machine processes, and/or distributed processes in any combination.

Parallelism in this algorithm is achieved by dividing the particle set, and hence the associated computation, amongst a collection of processes. As the following explains, the key is that in each simulation step, each process needs to see the current data of every other process exactly *once*: this is accomplished by forwarding the particle data of each process around a circular Worker pipeline as noalias messages.

***n*-Body protocols.** To form the circular pipeline, each agent (Worker) creates the link to its right-hand neighbour (Figure 6). This means each Worker is both a "client", with respect to the next Worker on the right, and a "server", to

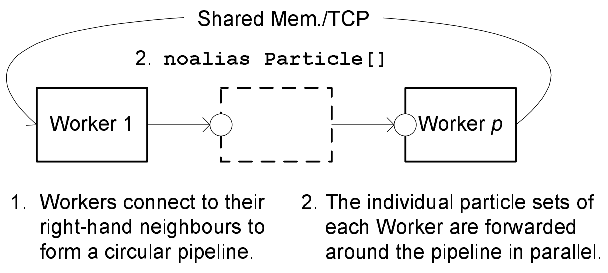
Shared Mem./TCP
2. noalias Particle[]
Worker 1
Worker p
1. Workers connect to their right-hand neighbours to form a circular pipeline.
2. The individual particle sets of each Worker are forwarded around the pipeline in parallel.

Figure 6. The p Workers in the n-Body pipeline.

the Worker on the left. However, once the pipeline formed, the interactions between each pair of Workers is the same. Following is the session type for these interactions from the server side of each Worker (the client side is dual type).

```
protocol workerServerSide {
  sbegin. // Accept link from left neighbour.
  !<int>. // Forward initialisation token.
  ?[ // Enter main simulation loop.
    ?[ // Inner iterations within each sim. step.
      ?(Particle[]) // Receive next particle set.
    ]*
  ]*
}
```

After accepting a session (`sbegin`) to the right-hand neighbour, the Worker forwards the initialisation token (an integer counter: `!<int>`). Then for each simulation step, as signalled by the left-hand neighbour (`?[`), the Worker enters the inner iterations (`?[`) that forward the particles data of each process anticlockwise (`?(Particle[])`) around the ring for all Workers to see. For this client-server design, the circular pipeline is bootstrapped by designating two neighbours to be the "first" and "last" pipeline units. This design works independently of the pipeline length ($p \geq 2$).

***n*-Body implementation.** In the SJ implementation of the above algorithm, each Worker opens a session server socket to accept the link from the left-hand neighbour,

```
ss_l = // SJServerSocket.
   SJServerSocket.create(workerServerSide, port);
... // Enter session-try scope for s_l.
  s_l = ss_l.accept();
```

and makes the link to the right-hand neighbour,

```
s_r = c_r.request();
```

where `c_r`, of session type `workerClientSide`, is a `SJService` [41] for the server socket of the right-hand neighbour. After the pipeline links are established, the number of Workers in the pipeline, p, is dynamically determined from the initialisation token. The Workers then enter the main simulation loop, listed in Figure 7. To keep the Workers synchronised with respect to the main simulation loop and the inner iterations within each simulation step, the control flow at each Worker is linked to both the left and right-hand neighbours through session-iteration chaining. As Figure 7 shows, both

```
noalias Particle[] current = ...;
s_r.outwhile(s_l.inwhile()) { // Session-iteration chaining: synch. with both n'bours at each sim. step.
  current = ...;              // Prepare our own particle data for sending.
  s_r.outwhile(s_l.inwhile()) { // Session-iteration chaining: inner iterations within each sim. step.
    ...                 // (i) Add the current data to the running calculation.
    s_r.send(current); // (ii) Forward the current data set (the 'current' variable becomes null).
    current = (Particle[]) s_l.receive(); // (iii) Receive the next particle data set.
  }
  ... // Calculate the final results for this simulation step and update our own particle data.
}
```

Figure 7. SJ implementation of the main simulation loop in each pipeline Worker for the parallel n-Body simulation.

the outer and inner loops are implemented by `s_r.outwhile` `(s_l.inwhile()){...}`, where `s_r` (resp., `s_l`) is the session socket for the link with the right-hand neighbour (resp., left-hand neighbour). Hence, once the pipeline is fired up by the "first" Worker, the Workers iterate in parallel following their left-hand neighbours.

To propagate the particle data of each Worker to all other Workers, each step of the simulation involves $p - 1$ inner iterations. In the first inner iteration, each Worker calculates the partial result from their own particle set, and sends their particle data to the right-hand neighbour. In the n-th iteration, each process adds the particle data set received in the previous iteration to the running calculation (marked (*i*) in Figure 7), forwards this data set on to the right-hand Worker (*ii*), and receives the next data set from the left-hand Worker (*iii*). The particle data originating from the right-hand neighbour is received by the end of the final inner iteration: each data set has now been seen by all Workers in the pipeline, allowing the final results for the current simulation step to be calculated. The `current` variable, which holds the current particle set received in each inner iteration, is declared noalias, enabling the particle data to be forwarded (*ii* and *iii*) via zero-copy transfer between shared memory Workers.

The n-body program demonstrates, beside the usage of noalias, how structured, type-safe session programming is ideally suited to capturing such interaction structures involving an arbitrary number of collaborating peers. As mentioned earlier, the p Workers can be deployed in any combination of co-process threads, co-machine processes, and/or distributed processes — relying on the SJ Runtime to dynamically negotiate the most suitable session transport (shared memory, pipes, TCP) in each link context — without requiring *any* modification of the source code. Such portability is becoming increasingly relevant with the rise in heterogeneous clusters and ad hoc networking environments, which necessitate new, high-level abstractions for handling diverse forms of concurrency, both inter-host and intra-host.

4. SJ for Distributed Applications

This section presents two larger size SJ applications, corresponding to ***Use cases 2 and 3*** described in §1. These applications demonstrate SJ session programming for wide area network domains, complementing the parallel algorithm examples in §3, which are primarily targeted at local area network (or single machine) environments.

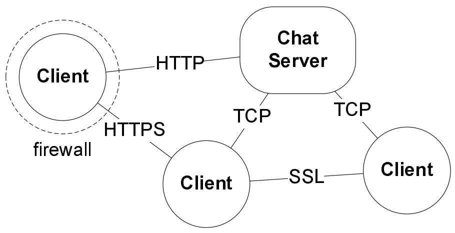

Figure 8. TCP, SSL and HTTP(S)-tunnelling in Chat.

The first is a typical Internet application, a chat server, that leverages SJ transport-independence between client-server (e.g. HTTP, for Web-based servers or firewall restrictions) and client-client (e.g. SSL) connections, and demonstrates SJ *service passing*, a complementary form of type-safe, higher-order communication to session delegation [14] which arises in many real-world protocols, such as FTP and HTTP-CONNECT. Chat also features session recursion. The second is a web-based corporate application portal. This application incorporates SJ code into Java applets and servlets, and demonstrates *transport-independent session delegation*, which enables session migration across differing transports.

4.1 Chat

Chat is a client-server application for real-time Internet text messaging. Like many Internet applications, it is sensitive to connection restrictions due to firewalls. The SJ Framework, however, decouples such concerns from the application itself: the SJ Runtime (SJR) includes a HTTP-based transport module for traversing firewalls via a Java servlet proxy (see §5), in addition to support for direct TCP connections.

In a simplified Chat (Figure 8), each Server maintains a single global conversation channel on which all connected Clients can read and write. Each Client also obtains from the Server a list of the other Clients connected to the Server, updated as Clients join and leave. A Client can use this information to request a private conversation with another peer: the request is made via the Server, but once estab-

```
protocol clientToServer {
  cbegin.    // Client requests a session.
  !<String>. // Send user name.
  ?(int).    // Receive user ID.
  ?(cbegin.@(eventInStream)). // (i).
  @(eventOutStream) // Start event out stream.
}

protocol serverToClient {
  ^(clientToServer) // Dual session type.
}

protocol eventInStream {
  ^(eventOutStream)
}
```

```
protocol eventOutStream {
  rec X[ // Enter recursion scope for stream.
    !{   // Select the event type.
      JOIN: !<UserJoinEvent>.#X,
      MESSAGE: !<MessageEvent>.#X,
      PRIVATE_CONVERSATION: // Make request and...
        !<PrivateConversationEvent>.?{ // ...get response.
          ACCEPT: !<cbegin.@(clientToClient)>, // (ii).
          REJECT:
        }.#X,
      QUIT: !<UserQuitEvent> // No recurse (stream end).
    }
  ]
}
```

Figure 9. Session types for the Client-Server interactions of a basic chat application featuring higher-order service passing.

lished, the private conversation is conducted completely separately from the global channel. The basic Chat implementation comprises approximately 4K SLOC, with 40 SLOC for protocol declarations and 600 SLOC of session code. We explain the session types that specify the Chat interactions and key implementation details below.

Chat protocol design. Chat comprises two main protocols. The first describes the interaction between Clients and Servers, as listed in Figure 9 from the Client perspective (`clientToServer`). `serverToClient` is its dual (`^` means "dual"), i.e. in (`?`) and out (`!`) qualifiers inverted. The bulk of a Client-Server session is given by the event stream protocols. `eventOutStream` allows the Client to signal the join and leave actions, send a message, and make a private conversation request (for brevity, most of the messages have been condensed into single `Event` types). `eventInStream` is the dual to `eventOutStream`. The second main protocol (`clientToClient`) is just a simpler version of the first that excludes the `JOIN` and `PRIVATE_CONVERSATION` branch cases for the private conversations between Clients.

The two key points are the declaration of the event streams using recursion and the higher-order *service passing* message types. Unlike the other session type constructors, recursion (`rec X[...#X]`) does not specify an interaction "direction". Session recursion binds local control flow to a separate, nested session construct, typically a branch element: in `eventOutStream`, only the `QUIT` case of the nested outbranch exits the recursion, terminating the stream.

The Client-Server interactions feature two instances of service passing. Each Client-Server session encapsulates a pair of dual event streams, to permit the Client and Server to operate with full asynchrony: either can send an event at any time. The first instance of service passing, marked (*i*) in `clientToServer`, specifies that the Server send the Client a `SJService` of type `cbegin.@(eventInStream)` where `@(eventInStream)` is the SJ notation for referencing `eventInStream`. The Client uses this service to initiate the Server-to-Client event stream (`eventInStream` from the Client perspective). The Client-to-Server event stream is commenced after (*i*) on the parent session. The second instance of session passing marked (*ii*) in `eventOutStream` occurs in the set-up for private conversations between Clients.

Chat implementation. We briefly outline the implementation of the key features described above, service passing and session recursion. After a connection and the preliminary exchange of user information (see `clientToServer` in Figure 9), the Server uses service passing (*i*) so the Client can initiate a Server-to-Client event stream. Server first opens a fresh port `port`, which is used to create an `SJService`.

```
SJService c = SJService.create(p_cs, serv, port)
```

where protocol `p_cs` is `cbegin.@(eventInStream)` (Figure 9) and `serv` is the address of the Server. Server then performs the service passing action by sending the `SJService` to Client:

```
s_sc.send(c); // !<cbegin.@(eventInStream)>
```

and binds a `SJServerSocket` to `port` of type `sbegin.@(eventOutStream)`, i.e. the dual session type to the type of `c`. Client receives the `SJService` (cast included for clarity) and makes the expected request (`params` specifies optional transport preference parameters):

```
SJService c =
  (cbegin.@(eventInStream)) s_cs.receive();
... = c.request(params); // Initiate in stream.
```

Server will `accept` the request in the usual way, and can then close the dynamic server socket and `port`.

There are two ways to implement a session recursion. One is the basic SJ session recursion construct, and the other is using *recursive session methods* [41] through a recursive method call, using method declaration for session implementation. We show the former for the Client-to-Server event out stream in the Client implementation:

```
protocol appletToPortal { // End-user applet.
  cbegin. // Request application session.
  ... // Preliminary exchange of parameters.
  !{
    WEBMAIL: @(appletToWebmail),
    CALENDAR: @(appletToCalendar),
    DATABASE: @(appletToDatabase),
    ...
  }
}
```

```
protocol portalToWebmail { // Portal connects to Server.
  cbegin.!<@(webmailToApplet)> // (iii).
}

protocol webmailToPortal { // Webmail is a Server.
  sbegin.?(@(webmailToApplet)) // (iv).
}

protocol webmailToApplet {
  ^(appletToWebmail)
}
```

Figure 10. Session types for the core interactions, including session delegation, for an intranet application portal.

```
s_cs.recursion(X) { // rec X[
  ...
  else if(...) { // Sending a message.
    s_cs.outbranch(MESSAGE) { // !{ MESSAGE:..
      s_cs.send(...); // !<MessageEvent>
      s_sc.recurse(X); // Recursive jump: #X
    } // }
  }
  else ...
} // ] Recursion scope exit.
```

As already discussed, transport independence gives Chat greater portability across differing communication contexts, such as traversing firewalls. As a further example, repackaging the Client application as a Webpage applet requires very little work: again, the SJR can transparently negotiate a HTTP-based session to connect to the parent Web-server. Enabling Chat Clients to conduct private conversations securely is a similar situation; due to SJ transport negotiation (see §5), the Client will automatically reject a private conversation request if either peer demands a secure transport that the other does not support. Further extensions to the Chat application that may introduce further transport requirements, such as voice communications and file transfer, would similarly benefit from development in the SJ Framework.

4.2 Application Server Portal

In this example, a developer presents a HTML embedded SJ applet as an Application Portal frontend to the various application servers hosted within the company network. The hosted applications could include a company calendar service, a Webmail client, and so on. On user request, the Portal applet establishes a HTTP-based session with the parent Web server from which applet originates. The parent server processes the user commands, and depending on the user requirements, *delegates* control (ownership) of the server-end of the session over to the appropriate application server. In addition to type-safety through higher-order session types, this example demonstrates the ability to perform transparent, *cross-transport session migration* using the SJ Framework. Although the applet communicates with the parent server using HTTP, session delegation can establish a direct TCP connection for better performance, or a secure session for handling sensitive data, with the application server (Figure 11). The key point is that the low-level and complicated machinery behind transport-independent delegation is cleanly abstracted away by high-level session programming, allowing the developer to focus on the application-level logic rather than the underlying transport and network details.

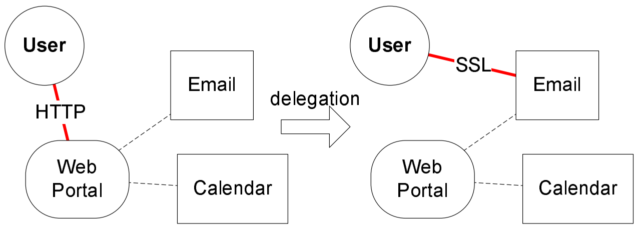

Figure 11. A Web-based portal (HTTP) to various application servers (TCP, SSL, etc.).

Application Portal protocols. Figure 10 lists the protocols between Applet, Portal and Servers, simplified to highlight the delegation actions at Portal. In the `appletToPortal` protocol, Applet selects from a range of applications, corresponding to the cases of the outbranch element (`!{...}`). The protocols between the Portal and each Server contain a single higher-order send type, the delegation. In `portalToWebmail`, for instance, the delegation marked (*iii*) has type `!<@(webmailToApplet)>`. This type describes that Portal delegates to Webmail the responsibility for completing the `webmailToApplet` session that Applet is expecting.

Application Portal implementation. The main SJ features of interest are the delegation in Portal and the corresponding session receive in each Server. At Portal:

```
noalias SJSocket s_pa; // To serve Applet.
... // Enter session-try scope for s_pa.
  s_pa = ss_p.accept(); // ss_p: SJServerSocket
```

Then depending on the user commands, Portal establishes a session to the target Server over which `s_pa` is delegated.

```
... if(...) { // Select application server.
  ... // Enter session-try scope for s_ps.
  s_ps = c_s.request(); // Connect to Server.
  s_ps.send(s_pa); // !<@(serverToApplet)>
}
```

where server in serverToApplet could be webmail, etc. For delegation, session typing requires s_pa to be noalias, which fits the concept of linear session delegation very naturally, i.e. Portal relinquishes this session (s_pa becomes null).

At the Server, the delegated Applet-Portal session is received by

```
... // Enter session-try scope for s_sa.
s_sa = (@(serverToApplet)) s_sp.receive(params);
... // Use s_sa to serve the Applet client.
```

where s_sp is the session socket for the Server-Portal session, and the optional params conveys transport preference parameters, as for request.

Transport-independent delegation. As discussed above, the end result of transport-independent delegation is the automatic negotiation of an available, and ideally the most suitable (based on runtime configuration and user preferences), transport for the delegated session. Firstly, this enables *cross-transport session migration*: although the Applet-Portal session may be HTTP-based, via the applet Web server, the application servers themselves may not support this transport. Secondly, in addition to the inherent performance benefits of reconnection-based delegation (as opposed to indefinite-forwarding [24], see §6), migrating the original session to, e.g. a direct TCP connection, increases performance potential, avoiding the need for polling by the applet (common in two-way HTTP-based communication) and reducing the load on the Web server. Thirdly, the mechanism underlying transport-independent delegation also provides the means to dynamically adopt/dispose of complementary session functionality, e.g. security through SSL/TLS, during the execution of a session.

5. SJ Runtime Design and Implementation

This section discusses the design and implementation of the SJ Runtime (SJR). We focus on the aspects of SJR architecture related to supporting the transport-independent execution of SJ programs for communication portability. The key design element is the *Abstract Transport*, which decouples the SJR *(session) interaction services* from the underlying transport mechanisms. In particular, we highlight how the Abstract Transport promotes flexible modularity and extensibility of both service and transport components in the SJR.

5.1 General Structure of the SJ Runtime

Figure 12 depicts the general structure of the SJR and some of the main components. Recalling the compilation and executions stages of a SJ Program (outlined in §2), the SJR has two main responsibilities. The first is to host the SJ *interaction services*. The SJ compiler translates session interaction in SJ programs into calls to these services. For example, the SJ code

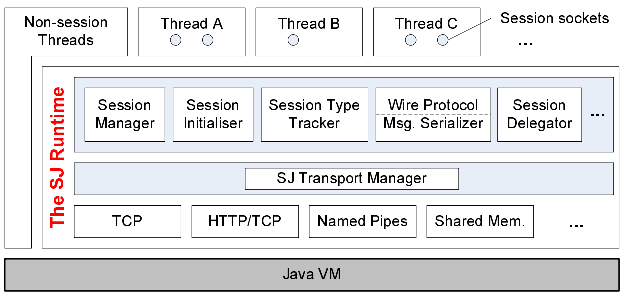

Figure 12. The structure and components of the SJR.

```
s.outwhile(...) {
  s.send(t); // t is of type noalias T.
  ... = s.receive();
}
```

is roughly translated to the Java in Figure 13. Services are interchangeably incorporated into the SJR as *service components*, which implement, for example, the protocols for initiation and delegation as well as the wire protocol and serialization format for communicating messages.

The second task of the SJR is the management of transport connections over which the interaction services are performed. Support for specific transports is incorporated through SJ *transport modules*, which implement the semantics of the Abstract Transport in terms of the communication mechanisms of the encapsulated "concrete" transport. The SJ Transport Manager (SJTM) uses the transport modules to open and close transport connections and perform the *transport negotiation* protocol (explained below).

Session sockets. In terms of session programming, the session socket is an endpoint abstraction for identifying sessions and implementing session interaction. At runtime, the SJR uses the corresponding SJSocket instances to record the state maintained for session execution. This state identifies the local and remote session connection endpoints (which uniquely identifies the session), the type of the requested session and the current progress, and optional transport configuration parameters. In addition, the SJR dynamically binds each active SJSocket with a handle to the underlying transport connection (explained further below). A SJSocket variable is implicitly (automatically) noalias: hence each thread conducting a session has complete ownership of the corresponding SJSocket (Figure 13). We also have a SJService which is a client-side end of a shared channel for session initiation, whereas a SJServerSocket serves as the server-side end of a channel for session initiation.

5.2 Abstract Transport

The Abstract Transport is a central device by which the interaction services are decoupled from concrete transport mechanisms. It represents the minimum sufficient functionality of a transport over which the session interaction services can be implemented. The Abstract Transport is in turn implemented

```
while (SJRuntime.outsync(s, ...)) {
  {
    T t_tmp = t;
    t = null;
    SJRuntime.pass(s, t_tmp);
  }
  ... = (...) SJRuntime.receive(s);
}
```

Figure 13. Calls to the SJR generated by the SJ compiler.

over concrete transports, permitting the session services to be performed over all compliant transports. For this purpose, the design of the Abstract Transport observes the following considerations.

Abstraction boundary. Provision of a clean abstraction boundary between the interaction service protocols and the properties of specific transports. This decoupling is achieved through the *Abstract Transport Interfaces* (ATI), which specify the functionality required from the underlying transports. The communication characteristics of the Abstract Transport are defined by the semantics of the ATI operations.

Portability. Transport independence from portability over diverse transports. A balance is required between the level of functionality offered by the Abstract Transport to the interaction services, and the ability to implement this functionality efficiently. We identify a minimal subset of the ATI as core operations to be implemented by all transports. The additional operations represent optional functionality (e.g. zero-copy transfer) for interaction services to exploit if available. If not directed supported, such functionality may be emulated within the Service Layer (e.g. timeouts).

The resulting design choices and their consequences are demonstrated below.

Communication characteristics. In the current SJ Framework, the Abstract Transport is based on asynchronous, reliable and order-preserving message delivery along the line of TCP and SCTP. This transport mode is one of the most widely used transport semantics in practice, and fits the practice [24] and semantics [14] of session-based programming, giving a simple and natural application-level semantics for high-level session interaction. These factors are crucial for effective portability of the Abstract Transport over diverse concrete transports.

While session programming under this communication characteristics can capture a wide range of applications (as the use cases in §3,4 demonstrate), the layered design of the SJ Framework readily allows a clean incorporation of other transport semantics. A treatment of alternative transport characteristics is discussed in §8.

Overview of the ATI. A compliant SJR *transport module* must implement the *core* ATI, which comprises the following three interfaces. A `SJConnection` represents an endpoint handle to an active Abstract Transport connection that supports the basic operations for reading and writing bytes. An important consideration is that the Abstract Transport permits asynchronous connection close. A correct implementation of `SJConnection` must allow the opposing endpoint to detect the close action on at least a read operation, which should return an I/O error. A `SJConnectionAcceptor` is an open session port that queues connection requests. Accepting a request returns the local `SJConnection` endpoint of the newly established connection. Lastly, `SJTransport` is the master interface that completes the presentation of the encapsulated transport as an implementation of the Abstract Transport. A `SJTransport` provides the means to create `SJConnectionAcceptor`s and make connection requests, and also specifies a mapping from SJ session-level port values to transport-specific entities (ports or otherwise).

Implementation of the core ATI provides sufficient functionality to perform the essential interaction services; hence, the SJR can be *rapidly* extended to incorporate additional transports. The ATI also include optional interfaces that indicate the encapsulated transport supports certain additional functionality to be exploited by the SJR and relevant interactions services. A good example is `SJLocalConnection`, which extends `SJConnection` with operations for zero-copy messaging transfer, which will be implemented by e.g. a shared memory transport (see below).

ATI implementation. We discuss the ATI implementation of some selected transports. The SJ Framework is designed to concentrate such low-level communication details within this layer, decoupling application-level interaction from the underlying transport mechanisms. This frees the programmer to focus on the application-level, whilst experts can develop additional and specialised transport modules.

TCP-like transports have similar properties to the Abstract Transport. Thus, creating a compliant transport module can take less than an hour, which is particularly useful for domain-specific tuning of such transports.

A **shared memory** transport module was completed in a few hours. The basic approach was to model the asynchronous (full-duplex) communication of the Abstract Transport using two FIFO buffers, one for each direction. This automatically gives reliability and order-preservation, provided access to the buffers by the two endpoints is correctly synchronised for blocking reads. Careful synchronisation is also needed to correctly "shutdown" both buffers and notify the opposing endpoint on connection close. The shared memory connection implements `SJLocalConnection` and performs zero-copy transfer by simply passing the message reference through the FIFO. Ordinary writes copy the data using serialization. The transport module internally manages "port" usage by active connections and acceptors.

HTTP transport module is more complicated, involving a couple of days work. There are two modes of operation: direct connection between the two endpoints, and connection via a servlet proxy. The current design aims to treats both modes as uniformly as possible. Session-level port values are mapped to TCP ports offset by an internal constant. A connection request first attempts a direct TCP connection to the specified target address. If unsuccessful, a connection to port 80 (configurable) on the target host on is attempted, sending a GET for the fixed resource path to the expected servlet. The servlet obtains the address of the intended connection target from the GET parameters and creates the proxy connection. After a connection is established, writes are performed using POST, with compliance for connection persistence. For direct connections, a read simply parses the first write POST request received from the opposing side. For proxy connections, a read is also performed using POST: the servlet parses the corresponding write from the opposing side and returns a response to the read POST.

5.3 Transport Manager

The SJR uses the SJTM to manage to open and close `SJConnection`s and connection *acceptor groups*. The SJTM uses the incorporated transport modules to open connections based on the two-phase *transport negotiation* protocol. As explained below, the concrete transport underlying a new `SJConnection` is determined from the validity of the available transports for connecting to the specified destination, and the transport preference parameters at *both* connection endpoints. The SJTM can additionally intercept a connection close, and cache the connection for later reuse.

As a rough description, a connection acceptor group collects together the `SJConnectionAcceptor`s for each transport supported by a session server socket. The acceptors for the transports for connection setup (explained below) are bound to the transport-level port given by the provided session-to-transport mapping. This permits a session server socket to encapsulate, e.g. multiple TCP-based transports, as long as the port mapping between the transports do not clash.

Transport negotiation. Creating a new `SJConnection` involves two phases. The first is to establish contact with the connection target by creating a preliminary *setup* connection between the requestor (**R**) and acceptor (**A**). The setup connection is then used to negotiate a transport for the pending *session* that both parties agree on. For this purpose, the SJR maintains separate preference lists for the setup (S) and session transports (T), which can be customised using the SJR configuration methods, and through the parameters to application-level session requests (`c.request(params);`) and session server socket creation. The transport negotiation protocol roughly proceeds as follows:

1. **R** attempts setup connections according to ordering of S (below s denotes the established setup transport).

2a. **R** sends a flag to indicate whether R1) negotiation is mandatory (s not in $T_\mathbf{R}$), R2) negotiation is unnecessary (s first priority in $T_\mathbf{R}$); or else R3) start negotiation.

2b. At the same time, **A** says either A1) no negotiation is possible (only s is in $T_\mathbf{A}$), A2) the setup transport is supported for sessions (s in $T_\mathbf{A}$), or A3) the server supports other session transports but not s (i.e. s is not in $T_\mathbf{A}$ and $T_\mathbf{A}$ is not empty).

This scheme permits both fast agreement and flexible negotiation. In the case of R2 or R3 paired with A1, the session should directly proceed over the already established setup connection; similarly for R2 and A2. The key is that both parties can independently determine this outcome from the initial exchange. In the case of further negotiation, **A** lists $T_\mathbf{A}$ and **R** can select a new transport or reuse the setup connection. R1 and A1 signals connection failure.

In the current SJR, commonly used setup transports are shared memory and TCP. However, it would be interesting to investigate alternative, lightweight methods for distributed transport negotiation, e.g. using UDP. The SJTM can additionally cache the transport preferences of known hosts, and whether previous connections using particular transports succeeded, to optimise the selection of setup transports. These optimisations "warm up" the SJR, akin to JIT techniques in the standard JRE.

Recall that the SJR directly binds connection handles to `SJSocket`s. Once a session is established, the interaction services can execute communication actions (e.g. send/receive) through direct direct access to the underlying connection via the session socket, incurring minimal cost, cf. §6.

5.4 Session Interaction Services

As illustrated in Figure 13, application-level session interaction is executed at runtime via calls to the interaction services, which carry out the interaction as Abstract Transport communications. The key point is that, just as the addition of a transport module immediately enables the session services to be performed over the new transport, replacing or adding service components to the SJR allows the new services to perform transparently over all transports. The following describes some of the key services, highlighting design elements related to transport-independence. Other Service Layer functionality includes session initiations, message serialization and wire protocol, and dynamic monitoring of messages (against the expected session types).

Noalias message passing. By default, the SJ compiler targets this service for all `pass` and `send` operations called with noalias arguments. If the target session is bound to a `SJLocalConnection`, this service uses the zero-copy transfer operations to pass the message. Otherwise, this service invokes the standard copy-on-send service. This is a simple demonstration of how services can transparently make the best use of the available transport functionality. Other such services include emulating input timeouts, described below.

Delegation. Session delegation requires intricate coordination between the parties involved. Alternative protocols for this coordination and their tradeoffs were studied in [24];

their implementations are incorporated into the SJR as delegation service components. The SJR is easily configured to utilise a specific protocol by interchanging the relevant service component. The reconnection-based delegation protocols can directly reuse the SJTM transport negotiation mechanisms to perform cross-transport session migration (§4.2), automatically fitting the delegated session to the most suitable transport (modulo user configuration) for the new execution context, cf. §6. Due to the Abstract Transport, adding a transport module to the SJR automatically extends this service to support session migrations to the new transport.

Input timeouts and non-blocking input. The read operation with timeout is part of the optional ATI in the current Abstract Transport. However, this function can also be realised as a *service*, giving a basic demonstration of how certain functionality can be emulated within the Service Layer if the underlying transport does not support it directly. The functionality can be achieved by spawning a worker thread to perform the blocking, non-timeout read, and starting the timer in the original thread. If a message is received before the timer expires, the worker can notify the original thread and return the message (or exception). Otherwise, the timer cleans up the worker and raises the timeout exception. Simple as it is, observe that this scheme enables input timeouts for all transports, including future ones.

Similarly a simple service, which internally uses a timeout as above, can realise a non-blocking input, which, if a session has already received a message, immediately returns one: and the null if not. This service can easily support asynchronous, event-based communication primitives in session programming, e.g. selecting one session out of multiple sessions based on the order of a message arrival (cf. [17]), under the same type-based safety assurance.

6. Benchmarks

This section presents performance measurements for the SJ Framework to demonstrate the feasibility and advantages of transport-independent design. The first benchmark compares the transport-independent SJ Framework against the preceding TCP-based version of SJ, which was shown to perform competitively against TCP sockets and RMI in [24]. The results show the new SJ Runtime, developed around the Abstract Transport, incurs minimal overheads in comparison with the TCP-specific implementation. We also use this benchmark to show the potential for significant performance gain due to noalias types. The second benchmark demonstrates the effectiveness of session delegation, particularly when the application can take advantage of cross-transport migration to more optimal transports.

Lastly, we present macro benchmarks featuring SJ implementations of larger applications with complex and representative interaction structures. A comparison of SJ with MPJ Express [1], a reference Java messaging system based on the MPI [31] standard, using the parallel algorithms in §3 yields further promising performance results for SJ. The results confirm that portability due to transport-independence allows real applications to (1) transparently make the best of available transports *without* code modification (§3.2), and (2) exploit noalias types for optimising linear communication patterns, such as exchanging ghost points (§3.1).

We used the same machines in the same network environment for the following benchmark experiments. Each machine is a dual-core Intel Core 2 Duo (E8400) at 3GHz with memory, running Ubuntu Linux 8.10 (kernel 2.6.27). The machines were connected via gigabit Ethernet, and the latency between two machines was measured using ping (64B) to be on average 0.10ms. The SJ compiler and Runtime and the benchmark applications were built and executed using the standard Sun Java SE compiler and runtime versions 1.6.0_12. The complete results and full benchmark source code can be found at [41].

6.1 Micro Benchmarks for Communication Overheads

Following the approaches taken in [29, 47], this set of micro benchmarks is based on the communication of binary trees, which represent a middle-ground in common data structures (between sparsely-connected lists and densely-connected graphs). Each benchmark was conducted for trees of increasing depth (0, 1, 2, 4, 8) in sessions of length (session-while iterations) 0, 1, 10, 100 and 1000. The SJR was configured to use shared memory and TCP for both connection setup and session execution; the old SJ is implicitly coupled to TCP. Nagle's algorithm was disabled for both. Each benchmark run was preceded by a full dummy run for stability, and we present the average results of 1000 repeats for each parameter combination.

Overhead of Abstract Transport. The aim of the first benchmark is to determine the overheads introduced by the ATI due to the decoupling of interaction services from transport modules. We compare the performance of the new SJR implementation to the preceding version of SJ [24] which is directly coupled to TCP without Abstract Transport. Taking this communication-dominated micro benchmark in a very low latency environment emphasises the internal overheads incurred by the SJR for session execution, whilst minimising the influence of other external performance factors; both implementations use a similar wire format and serialization protocol. Since the previous SJ was measured against TCP and RMI, we can also compare the overheads against them.

We measure the time taken for the Client to complete the following session with the Server after the session has been established. The Client and Server are run on separate machines, so the SJR negotiates TCP for the session. The protocol from the Client side is

```
(1)   cbegin.![!<BinTree>.?(BinTree)]*
```

and the Server implements the direct dual type. The Client and Server repeatedly exchange (and update) a binary tree for as long as the Client continues the iteration (session

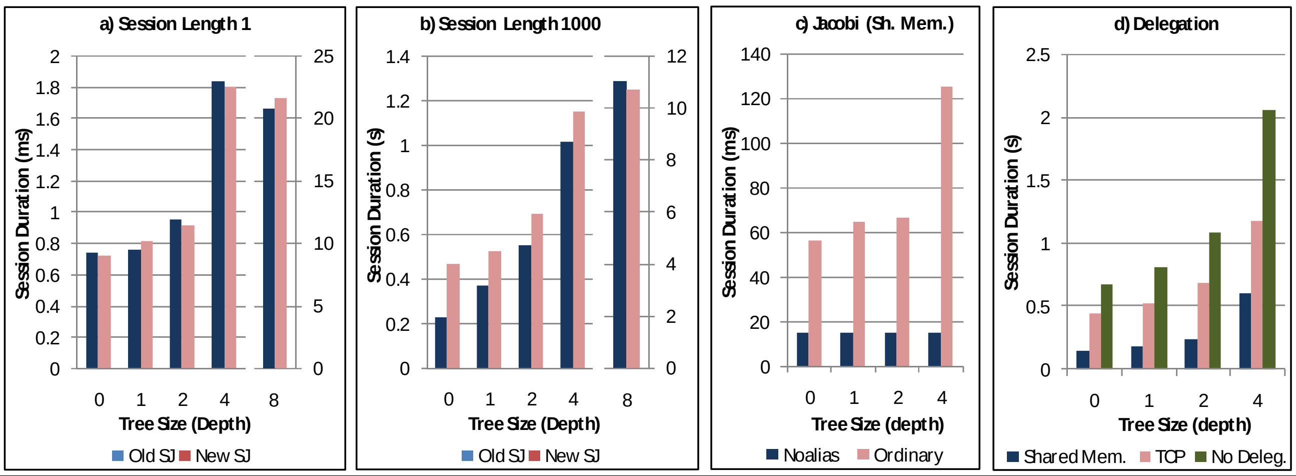

Figure 14. a) and b) new SJR vs. old TCP-based SJ; c) shared memory sessions with ordinary and noalias message passing; d) (cross-transport) session migration from TCP to TCP and TCP to shared memory against non-delegation base case.

length). In addition to these messages, recall that session-iteration involves implicit communication (performed by an interaction service) between the peers.

Figure 14 a) and b) show for session lengths 1 and 1000 (as representatives of short and long sessions), the mean time (y-axis) to complete the session for the varying tree sizes (x-axis). These results show that the new SJ Framework with Abstract Transport incurs very little additional overhead in comparison to the preceding TCP-based version. For the shorter session, the difference in performance is minimal. The overheads are more visible for the longer session, but still confirm the feasibility of our design.

Noalias types. To illustrate the impact of noalias types for transparent zero-copy message transfer, we reuse the first benchmark to run Client and Server as co-VM threads (shared memory); this requires no changes to the original source code, although a few lines of external code are used to bootstrap the threads. We compare this multithreaded "ordinary" version against a version that uses noalias `BinTree`s.

Figure 14 gives the time to complete sessions of length 1000 for trees of depth 0, 1, 2 and 4. These results indeed show that exploiting noalias types offers the potential for greatly improved performance. For the version noalias, the communication costs, i.e. passing messages by reference, remain constant regardless of message size, whereas the cost of copy-on-send (via serialization) naturally increases with the tree size. The results for trees of depth 8 (18ms for noalias, 1020ms for ordinary) confirm the above, but are omitted from the graph for legibility. We also measured that the overheads introduced by noalias types for TCP sessions (i.e. zero-copy is not possible) are negligible, about 0.1%.

6.2 Cross-Transport Session Migration

This benchmark set demonstrates how transport-independence augments the advantages of reconnection-based session delegation [24] versus the emulation of delegation via indefinite forwarding. The common starting point for the benchmark application is for Alice and Carol to establish the session (a), and Bob and Carol the session (b). However, the bulk of the communication in this application consists of the messages passed between Alice and Bob (1000 `BinTree` exchanges). This is a generalisation of both the Chat (the private Client-Client conversations) and Application Portal (Applet-Server application session) scenarios in §4. We measured the time taken by Alice to complete her side of the session after (a) is established for the following three configurations.

Control case. In the base case, Alice and Bob are clients to the server Carol, each on a separate machine. The types of (a) and (b) are both (1) as specified above from the perspectives of Alice and Bob. Alice and Bob communicate using Carol as an intermediary, who manually forwards on the Alice-Bob messages in both directions. This scheme is similar to the mechanism of Mobile IP [25].

Delegation over TCP. Here, rather than relaying the messages between Alice and Bob, Carol delegates (a) over (b) to Bob. The type of (b) is now

(2) `sbegin.!<?[?(BinTree).!<BinTree>]*>`

from Carol's perspective. After the delegation, Alice and Bob perform the session communication over a direct TCP connection, although the delegation is transparent to Alice at the application-level (the type of (a) is unchanged).

Session migration from TCP to shared memory. The delegation-based implementation was repeated with Alice and Bob as co-VM threads. This means (a) and (b) are established over TCP as above, but the SJR transparently migrates the delegated session to a shared memory connection between Alice and Bob. Similarly if Alice and Bob were co-machine processes (session migration to e.g. pipes) or machines in a HPC cluster (e.g. RDMA).

(a) Size	Time (ms) Ordinary	Noalias
100	1270	992
300	24436	19448

(b) Size	Time (ms) SJ	MPJ
100	3713	4460
300	19501	19834

Figure 15. Jacobi: (a) "ordinary" vs. noalias versions (shared memory); (b) SJ vs. MPJ Express (distributed).

Results. Figure 14 gives the results for the delegation benchmarks for trees of depth 0, 1, 2 and 4 (depth 8 continues the trend but is omitted for legibility). As expected, the control case is considerably slower due to the overheads of redirecting the Alice-Bob messages via Carol; reconnection-based delegation benefits from dynamically reconfiguring communication topology to better reflect the evolving structure of the distributed application. The cross-transport migration to shared memory in turn improves on the TCP delegation, demonstrating the ability of the SJR to transparently migrate sessions to more optimal transports.

6.3 Macro Benchmarks: Parallel Algorithms

We present performance measurements for the parallel algorithms from §3. The first benchmark shows that the SJ Runtime, although an early implementation version with much scope for further optimisation, can perform competitively with MPJ Express [1]. MPJ Express adopts a pure Java approach[2]which makes for interesting comparison with SJ (as opposed to JNI wrapper libraries to C functions). These benchmarks were conducted in the same environment as above but using different machines: dual-core Intel Core 2 Duo (Conroe B2) at 2.13GHz with 2M L2 cache, 2GB main memory, running Ubuntu Linux 4.2.3 (kernel 2.6.24).

Jacobi Poisson solution. The parallel Jacobi algorithm is used to demonstrate (1) the effectiveness of SJnoalias types for larger, more complex applications, and (2) compares SJ performance to MPJ Express. Firstly, "ordinary" (i.e. without noalias) and noalias versions of the Master and two Workers were run as co-VM threads on a single machine. We measured the time to complete the algorithm for square matrices of size (the length of one side of the matrix) 100 and 300. In both cases, the noalias version is approximately 20% faster than the ordinary one (Figure 15a). Secondly, the SJ Jacobi, running each process on a separate machine (to use TCP, because MPJ Express does not support zero-copy in shared memory) was compared to an equivalent MPJ Express implementation: the SJ version performs better than MPJ Express by 6% on average (Figure 15b). We also implemented an extended version of SJ Jacobi which uses a linear pipeline topology, rather than multicast session-iteration, to support a dynamic number of Workers. This version performed on average 20% slower than the above version with multicast iteration. Thus this new multicast primitive can improve performance.

[2] Micro benchmarks comparing MPJ Express to other messaging systems are presented in [37, Chapter 8]. Their results show package transfer time in MPJ Express is slower than MPICH by an average 20% over fast Ethernet.

Num. of Particles	Mean execution time (ms) Threads	Localhost	Distributed
100	326	452	496
300	865	1144	1194
1000	6785	7497	7702

Figure 16. n-Body simulation: Threads vs. Localhost vs. Distributed versions.

n-Body simulation. The parallel n-Body algorithm features a complex collaboration pattern between a dynamic number of session peers, implemented using session-iteration chaining. We use this more advanced example (without noalias types, for a direct transport comparison) to concretely test portability due to transport-independence, in particular for flexible application deployment. The same implementation without source code modification was used to run a two Worker simulation (100, 300 and 1000 particles each Worker) in the following three configurations: as co-VM threads using shared memory (Threads), as separate processes on the same machine using TCP-loopback (Localhost), and on separate machines using TCP (Distributed).

As expected, the results (Figure 16b) show the Threads version is faster than Localhost: around 27% for 100 particles, 24% for 300, and 10% for 1000. The Localhost version is in turn slightly faster (latency is very low) than Distributed: 9% for 100 particles, 4% for 300, and 3% for 1000. The relative performance gain between each version decreases for larger particle sets because the local computation costs begin to dominate the communication costs for this fixed number of Workers. Naturally, performance can be improved for simulations of large particle sets by increasing the degree of parallelism, i.e. using more Workers.

7. Related Work

Message-based parallel programming. MPI [31] is one of the most widely-used APIs for parallel programming using message passing. Implementations supply concrete language and transport bindings, such as C, C++ and Fortran over one or more specific transports. The present work focuses on the integrated language-runtime design for transport-independent communications programming, rather than a supplementary API. In comparison to the standard MPI libraries [20, §4], the SJ Framework offers benefits in productivity coming from natural abstraction of communication actions by typed sessions as well as associated static assurance of type and protocol safety, as we discussed in §3. The MPI API remains low-level, easily leading to synchronisation errors, message type errors and deadlocks [20]. From our experiences, we found programming these and other message-based parallel algorithms in the SJ Framework through typed sessions based on Java much easier than programming based on the basic MPI functions, which, beside lacking in type checking for protocol/communication safety, often requires manipulating numerical process identifiers and array indexes (e.g. for message lengths in the n-body program) in tricky ways. On the other hand, the SJ Framework integrates objects and sessions to support, e.g. Java objects as high-level messages types with the facility for remote class loading [24]. SJ is also able to exploit session types to solve several MPI problems: a session is inherently communication-safe and deadlock-free [22]. Further, the benchmark results in §6 demonstrate how SJ programs can exploit transport-independent, type-directed optimisations to outperform a Java-based MPI implementation [1].

OpenMP and PGAS Languages. OpenMP [33] is a combination of pragma-based program transformation and libraries for extracting latent parallelism (shared memory multithreading) from sequential code. X10 [49], Chapel [12] and Fortress [18] are recent PGAS languages for HPC. These libraries and languages focus on reducing programming complexity for shared memory parallelism through a range of annotations and high-level constructs for coordinating and synchronising thread behaviours. The SJ Framework enables the abstraction and control of a collection of data transfer as a single session, leading to an optimal transport usage, and safety assurance through static type checking.

Multi-transport runtimes. APIs for object-based communications such as Java RMI and CORBA are typically tied to a few specific transports. Motivated by the need for domain-specific transports in distributed real-time and embedded systems, the OMG Extensible Transport Framework (ETF) [32] was developed to specify a transport plug-in framework for CORBA platforms. The ETF prescribes a set of interfaces to be implemented by the plug-in components in order for ORBs to communicate via alternative transport modes. The ATI and ETF interfaces define similar basic operations for connection open and close and byte-level read and write. However, certain aspects of the ETF are implicitly tied to the GIOP, making it difficult to, e.g. employ different marshalling schemes or extend communication beyond binary method calls [8]. Amongst other features, SJ supports transparent, cross-transport session migration, which CORBA/ETF has no facility for.

JXTA [27] is a platform for P2P applications, providing basic P2P protocols such as peer discovery and advertisement over multiple protocols (e.g. pipes can be implemented over differing concrete protocols). JXTA abstracts concrete transports to offer APIs for high-level P2P interactions so that the application development over JXTA is insulated from the low-level implementations involving concrete transports. Similarly messaging middlewares [3, 46] which support multiple transports are restricted to domain-specific functions and APIs, without general programming support, while the SJ Framework fully exploits transport-independence, advocating general typed communications programming supported by extensible runtime.

Distributed objects. Emerald [7] is an object-oriented language and system which supports object mobility in a network environment: it integrates a system which includes typed objects, their distribution-aware runtime environment, and fine-grain (object-granularity) distribution transparent to calls by other objects. It presents a fully-fledged design of distributed objects and their implementation framework including fine-grain object migration, while retaining the semantics of method call for both local and remote communications. The aim of the SJ Framework is quite different, investigating the integration of expressive, type-safe communications programming and object-orientation, through the enrichment by typed sessions – in particular, session migration is different from object and process migration. As a means for communications programming, the session abstraction has several advantages over the fixed shape of a synchronous method call.

Programming abstractions for communications. One of the themes of the SJ Framework is the provision of effective typed abstractions for concurrent and distributed programming. Polyphonic C♯ [4] offers a typed language with a join-based concurrency primitive [19] on the basis of C♯. The integration of the actor model with Scala is studied in [21]. In the context of functional languages, CML [35] offers a collection of elegant primitives centring on the higher-order abstraction *event*. Recently the notion of events is extended by adding a transactional property, called *transactional events* [15], so that they satisfy standard isolation properties with respect to communications and state change [16]. FoxNet [6] studies typed and modular implementations of real-world network stacks using standard ML's module abstractions, resulting in safe and competitive code base. Occam-π [48] is a system-level concurrent language based on CSP and the π-calculus, offering highly efficient channel-based synchronous communication primitives as well as various locking and barrier abstractions.

Unlike the SJ Framework, these works do not (intend to) provide typed programming abstractions which assure protocol safety for communications programming, nor do they investigate the programming and runtime support for transport independence. An integrated understanding of the interplay among these and other typed concurrency abstractions will be an interesting topic for further study.

Zero-copy communication with linear types. The noalias mode in SJ is similar to the `unique` annotation of [2] for describing unshared references. Our contributions include demonstrating how the integration of object alias control and message passing programming can have significant performance benefits whilst retaining semantic transparency. SJ differs from [2] by directly capturing the semantics of noalias assignment and argument passing operationally, rather than enforcing particular patterns for variable usage. The above design choice of SJ, in contrast to [2], dispenses with manual synchronization for field accesses and leads to a different inference of noalias-compatible classes.

Sing#, a derivative of C# developed for Singularity OS [17], uses a variant of session types called *contracts* to specify the interfaces between OS components, which communicate via channel-based message passing in shared memory environments. Their approach is based on ownership transfers to support message exchange via a specially designated heap area in shared memory. Kilim [40] is an actor framework for Java based on cooperatively-scheduled lightweight threads which communicate by message-passing. By static checking, a message can have at most one owner at any time, allowing efficient zero-copy transfer. StreamFlex [39] is a real-time stream API for Java guaranteeing sub-millisecond response times and type safety. It makes use of a type-based classification of heap objects which in effect ensures linearity of messages carried through streams, leading to a high throughput. The typing disciplines in these preceding works are however designed for specifically restricted objects for communication distinct from general objects.

The noalias type in SJ is a general mode declaration for standard alias control, not restricted to the use of zero-copy message passing in shared memory environments. Thus our framework permits the same communication mechanisms at the language-level with alias type checking, whilst uniformly handling linearity of session communication in a transport-independent manner. This means zero-copy or other delivery mechanisms can be exploited in SJ for any compliant transports without altering semantics, whereas object transfer in the above work are tightly coupled with their transport mechanisms.

While the aim of each work is different, controlling behaviour by linear types for efficiency and safety is common in these and our work: we hope the efforts based on previous study of linear types, e.g. [17, 40, 45], can be applied to both the safety assurance and performance of concurrency and communication in programming languages.

Implementations of session types. A framework based on F# for cryptographically protecting execution of sessions from external attackers and malicious principals is studied in [5, 13]. Their session specification graphs are used for modelling and validating the integrity of communication sequences between two or more network peers. Their work focuses more on ensuring interaction properties, not the realisation of the interaction over specific or transport-independent mechanisms. The protocols for SJ session initiation can be easily extended to incorporate security mechanisms, such as peer authentication using certificate exchange and key negotiation for encryption, on the basis of the current SJ implementation of SSL and HTTPS. The above works have not investigated a language-runtime support for the range of features integrated in the SJ Framework, such as transport independent, type-directed optimisations in concurrent and distributed environments.

8. Conclusions

This paper has argued for the significance of portability in communications programming through *transport independence*. Our new language-runtime framework recognises the need for both suitable language facility and runtime support to take full advantage of transport-independent programming. The aim is not only to allow such programs to make the best use of diverse transports, but to facilitate the task of communications programming for the diverse application domains that require these transports. The framework hinges on the session-based abstraction for communications, complementing object-oriented abstraction for local computation, offering accurate and flexible programming abstractions for structured conversations, assuring type and protocol safety, and functioning as a link between high-level transport-independent abstraction and runtime execution. Transport-independence stems from the decoupling of application-level interaction semantics from lower-level communication mechanisms, linked by session abstraction.

We substantiated the significance of transport independence in programming abstraction and execution, through the implementations of several concrete applications with complex session interactions based on use cases in differing domains, from algorithms to a large size of distributed applications. The parallel algorithms in the SJ Framework have clear, readable communication structures with protocol safety ensured by session type checking, and are inherently portable across transports available in e.g. SMP and clusters, and transparently exploit zero-copy transfer in shared memory. Distributed applications, including the Internet chat and application server portal, demonstrate how the SJ Framework enables the opportunistic use of multiple transports in the present-day network environments, including HTTP for firewall traversal and Web-based deployment, SSL for secure communications, and TCP, without needing any source code modification. The application server portal also demonstrates the use of transparent cross-transport session migration. These applications show the SJ Framework can have considerable impact on productivity whilst enjoying type-safety, deadlock-freedom and competitive performance.

The current SJ Framework is a basis of the investigation of many further topics. Firstly, making full use of SJ Runtime extensibility, we may integrate not only further transports (e.g. RDMA for HP clusters) but also alternative transport modes, such as transports with order-preserving lossy message delivery [28] or unambiguous unordered delivery [3]. Secondly, the extensibility also applies to the addition of new session interaction services to those extensions discussed in §5 for enriching the facility of session programming: possible additions include session hibernation, failure recovery, and process migration [38], using the protocols closely related to session delegation [24]. These services will also result from extensions to session types, e.g. distributed exceptions [10], and multiparty extensions [9, 23]. Runtime session type information can be exploited for many purposes including user-level reflection, security and debugging, as well as more fine-grained optimisation for e.g. serialization. Session programming may also benefit from type-inference for session and noalias types, combining the methods developed in [2, 30].

As industry-scale applications of the SJ Framework, we are currently experimenting large distributed application frameworks in collaboration with our industry colleagues, through the use of the international standards for financial and business protocols, WS-CDL and UNIFI [11, 44].

References

[1] MPJ Express homepage. `http://mpj-express.org/`.

[2] J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *OOPSLA*, pages 311–330, 2002.

[3] Advanced Message Queuing protocols (AMQP) homepage. `http://jira.amqp.org/confluence/display/AMQP/Advanced+Message+Queuing+Protocol`.

[4] N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C#. *TOPLS*, 26(5):769–804, 2004.

[5] K. Bhargavan et al. Cryptographic Protocol Synthesis and Verification for Multiparty Sessions, 2008. Draft.

[6] E. Biagioni, R. Harper, and P. Lee. A network protocol stack in Standard ML. *HOSC*, 14(4):309–356, 2001.

[7] A. P. Black, N. C. Hutchinson, E. Jul, and H. M. Levy. The development of the emerald programming language. In *HOPL III*, pages 11–1–11–51. ACM, 2007.

[8] D. Borusch1 et al. Integrating the romiop and etf specifications for atomic multicast in corba. In *ESOP*, volume 3760 of *LNCS*, pages 680–697. Springer, 2005.

[9] M. Carbone, K. Honda, and N. Yoshida. Structured Communication-Centred Programming for Web Services. In *ESOP'07*, volume 4421 of *LNCS*, pages 2–17. Springer, 2007.

[10] M. Carbone, K. Honda, and N. Yoshida. Structured interactional exceptions in session types. In *CONCUR*, volume 5201 of *LNCS*, pages 402–417. Springer, 2008.

[11] W3C Web Services Choreography. `http://www.w3.org/2002/ws/chor/`.

[12] Chapel homepage. `http://chapel.cs.washington.edu/`.

[13] R. Corin, P.-M. Denielou, C. Fournet, K. Bhargavan, and J. Leifer. Secure Implementations for Typed Session Abstractions. In *CFS'07*. IEEE-CS Press, 2007.

[14] M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida, and S. Drossopoulou. Session Types for Object-Oriented Languages. In *ECOOP'06*, volume 4067 of *LNCS*, pages 328–352. Springer, 2006.

[15] K. Donnelly and M. Fluet. Transactional events. *J. Funct. Program.*, 18(5-6):649–706, 2008.

[16] L. Effinger-Dean, M. Kehrt, and D. Grossman. Transactional events for ML. In *ICFP*, pages 103–114. ACM, 2008.

[17] M. Fähndrich et al. Language Support for Fast and Reliable Message-based Communication in Singularity OS. In W. Zwaenepoel, editor, *EuroSys2006*, ACM SIGOPS, pages 177–190. ACM Press, 2006.

[18] Fortress homepage. `http://projectfortress.sun.com/Projects/Community`.

[19] C. Fournet, C. Laneve, L. Maranget, and D. Rémy. Inheritance in the join calculus. *J. Log. Algebr. Program.*, 57(1-2):23–69, 2003.

[20] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI.* MIT, 1999.

[21] P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *TCS*, 410(2-3):202–220, 2009.

[22] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 22–138. Springer, 1998.

[23] K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In *POPL'08*, pages 273–284. ACM, 2008.

[24] R. Hu, N. Yoshida, and K. Honda. Session-Based Distributed Programming in Java. In *ECOOP'08*, volume 5142 of *LNCS*, pages 516–541. Springer, 2008.

[25] IETF. Mobility for IPv4. `http://dret.net/rfc-index/reference/RFC3344`.

[26] Java Message Service (JMS) homepage. `http://java.sun.com/products/jms/`.

[27] JXTA homepage. `https://jxta.dev.java.net/`.

[28] E. Kohler, M. Handley, and S. Floyd. Designing DCCP: congestion control without reliability. *SIGCOMM Comput. Commun. Rev.*, 36(4):27–38, 2006.

[29] J. Maassen et al. Efficient Java RMI for parallel programming. *ACM TOPLAS*, 23:747–775, 2001.

[30] L. Mezzina. How to infer finite session types in a calculus of services and sessions. In *COORDINATION'08*, volume 5052 of *LNCS*, pages 216–231. Springer, 2008.

[31] Message Passing Interface (MPI). `http://www-unix.mcs.anl.gov/mpi/usingmpi/examples/intermediate/main.htm`.

[32] Common object request broker architecture: Core specification. OMG document formal/04-03-01, 2004.

[33] OpenMP homepage. `http://openmp.org/wp/`.

[34] Polyglot homepage. `http://www.cs.cornell.edu/Projects/polyglot/`.

[35] J. H. Reppy. CML: A higher-order concurrent language. In *PLDI*, pages 293–305, 1991.

[36] J. Saltzer, D. Reed, and D. Clark. End-to-end arguments in system design. *ACM Transactions in Computer Systems*, 2(4):277–288, 1984.

[37] A. Shafi. *Nested Parallelism for Multi-core Systems Using Java*. PhD thesis, University Of Portsmouth, 2006.

[38] A. C. Snoeren and H. Balakrishnan. An end-to-end approach to host mobility. In *MOBICOM*, pages 155–166, 2000.

[39] J. H. Spring, J. Privat, R. Guerraoui, and J. Vitek. StreamFlex: high-throughput stream programming in Java. In *OOPSLA*, pages 211–228. ACM, 2007.

[40] S. Srinivasan and A. Mycroft. Kilim: Isolation-typed actors for Java. In *ECOOP*, volume 5142 of *LNCS*, pages 104–128. Springer, 2008.

[41] SJ homepage. `http://www.doc.ic.ac.uk/~rh105/sessionj.html`.

[42] K. Takeuchi, K. Honda, and M. Kubo. An Interaction-based Language and its Typing System. In *PARLE'94*, volume 817 of *LNCS*, pages 398–413, 1994.

[43] A. S. Tannenbaum. *Computer Networks*. Prentice Hall, 1996.

[44] UNIFI. International Organization for Standardization ISO 20022 UNIversal Financial Industry message scheme. `http://www.iso20022.org`, 2002.

[45] D. Walker. *Advanced Topics in Types and Programming Languages*, chapter Substructural Type Systems. MIT Press, 2005. Editor Benjamin C. Pierce.

[46] WebSphere homepage. `http://www-01.ibm.com/software/websphere/`.

[47] M. Wegiel and C. Krintz. Xmem: type-safe, transparent, shared memory for cross-runtime communication and coordination. In *PLDI*, pages 327–338. ACM, 2008.

[48] P. Welch and F. Barnes. Communicating Mobile Processes: introducing occam-pi. In *25 Years of CSP*, volume 3525 of *LNCS*, pages 175–210. Springer, 2005.

[49] X10 homepage. `http://x10.sf.net`.